Garbage Collection

Algorithms for Automatic Dynamic Memory Management

Richard Jones

University of Kent at Canterbury, UK

Rafael Lins

Universidade Federal de Pernambuco, Brazil

JOHN WILEY & SONS

Chichester · New York · Weinheim · Brisbane · Toronto · Singapore

Baffins Lane, Chichester,
West Sussex PO19 1UD, England
National 01243 779777
International (+44) 1243 779777
e-mail (for orders and customer service enquiries): cs-books@wiley.co.uk
Visit our Home Page on http://www.wiley.co.uk
or http://www.wiley.com

Reprinted February 1997
Reprinted November 1997
Reprinted January 1999

Other Wiley Editorial Offices

John Wiley & Sons, Inc., 605 Third Avenue,
New York, NY 10158-0012, USA

Weinheim. Brisbane. Toronto. Singapore.

Library of Congress Cataloging-in-Publication Data
Jones, Richard, 1954–
Garbage collection : algorithms for automatic dynamic memory management / Richard Jones and Rafael Lins.
p. cm.
Includes bibliographical references and index.
ISBN 0 471 94148 4 (alk. paper)
1. Garbage collection (Computer sciences) 2. Memory management (Computer sciences) 3. Computer algorithms. I. Lins, Rafael.
II. Title.
QA76.9.G37J66 1996
005.4'2—dc20 96–14901
CIP

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

ISBN 0 471 94148 4

Typeset in 10/12pt Times by Richard Jones
Printed and bound in Great Britain by Bookcraft (Bath) Ltd
This book is printed on acid-free paper responsibly manufactured from sustainable forestation, for which at least two trees are planted for each one used for paper production.

7.5 Inter-generational pointers

Generational garbage collection reduces pause times by collecting only a region of the heap rather than its entirety. However, the only reference to an object in this region may reside in an area of the heap outside the region. It is vital that these inter-generational pointers be identified so that they can be treated as part of the root set by the scavenger. This can be done by the mutator, the garbage collector or a combination of the two.

The simplest way to find inter-generational pointers would be to scan older generations at collection time. The advantage of this method is that it can be done at no cost to the mutator, but it requires more scanning and has worse locality than a fully generational collector. However, linear scanning is faster than tracing and has better locality. Studies by Shaw and Swanson suggest that this technique can reduce overheads due to garbage collection by nearly a third compared with a completely non-generational two-space copying collector [Swanson, 1986; Shaw, 1988]. Bartlett uses a similar technique for his conservative garbage collector [Bartlett, 1989a]. The collector conservatively marks immediately reachable objects (i.e. global variables, references held in registers and on the stack), and then these objects are searched for inter-generational pointers. In this section, we consider more precise methods of recording inter-generational pointers.

The write-barrier

Pointers into a generation generally arise in two ways, either through pointer stores or through promotion of objects that contain pointers[8]. The latter are easily detected by the scavenger. If scanning older generations is ruled out, then pointer stores must be trapped and recorded. Barriers can be implemented in several ways, by either hardware or software, or with operating system support. Software barriers can be provided by having the compiler emit a few instructions before each pointer read or write. Hardware techniques do not require additional instructions, and so are especially advantageous in the presence of uncooperative compilers. Although hardware methods give the least mutator overhead, they may require special purpose hardware or modifications to the virtual memory system not generally available.

If software techniques are used, the implementor must consider three factors: how the cost to the mutator can be minimised, the space overhead of recording pointer store and how efficiently old–young pointers can be identified at scavenge time. If the barrier is sufficiently simple it can be compiled inline. However, pointer accesses may be very common, particularly in functional and object-oriented languages. Zorn found that the static frequencies of pointer loads and stores in SPUR Lisp were 13 to 15 percent and 4 percent respectively [Zorn, 1990a]. Inlining barriers may cause the size of the code generated to explode. If code expansion is sufficiently small (less than 30 percent) it may have negligible effect on performance provided that the processor's instruction cache is sufficiently large [Steenkiste, 1989].

An alternative is to use the operating system's virtual memory protection mechanisms,

[8] Inter-generational pointers also arise in system-specific circumstances. For example, a generational system for Prolog using the WAM might start a new generation whenever a new choice point is set. Old–young pointers have to be recorded by Prolog's unification algorithm since they must be reset on backtracking [Appleby *et al.*, 1988].

either to trap access to protected pages, or to use the page modification dirty bits as a map of the locations of cells that might have had pointer fields updated. The advantage of using virtual memory is that it is portable, requiring no changes to the compiler. However, Zorn's measurements suggest that its performance may be substantially inferior to software methods, although different architectures and operating systems vary considerably [Zorn, 1990a].

Fortunately it is not necessary to trap all stores. The proportion of stores that have to be trapped can be reduced by compile-time analysis. Stores to registers or to the stack need not be trapped if these locations are part of the root set of every garbage collection. Many stores, for example that of Lisp's *cons* operator, are initialising stores. As such, they cannot point forward in time so need not be trapped. Fortunately these two cases form the great majority of pointer stores. Zorn estimated that only 5 to 10 percent of all memory references were non-initialising pointer stores, and that two-thirds of these were writes to objects in the youngest generation [Zorn, 1990a]. Not all languages can readily recognise initialising stores, however. Many imperative languages separate the *allocation* of a heap object (for example, `x=malloc(...);` in C) from its *initialisation* (for example, `x->p=...;`). Even though only 1 percent of instructions generated by Lisp or ML compilers may be non-initialising pointer stores, optimising the write barrier is critical for overall performance. For example, if the write-barrier were to add 10 instructions to each of these stores, overall performance would be diminished by ten percent. We now consider methods of trapping and recording inter-generational pointers.

Entry tables

The first generational collector, by Lieberman and Hewitt, arranged for pointers from older generations only to point indirectly to objects in younger generations [Lieberman and Hewitt, 1983]. Each generation had an *entry table* of references from older generations associated with it (see Diagram 7.17). Whenever a pointer to a younger generation object was to be stored in an older generation object, a new entry was added to the younger generation's entry table, pointing to the young object, and the old object was modified to point to this entry. If the old object already contained a reference to an item in an entry table, that entry was removed.

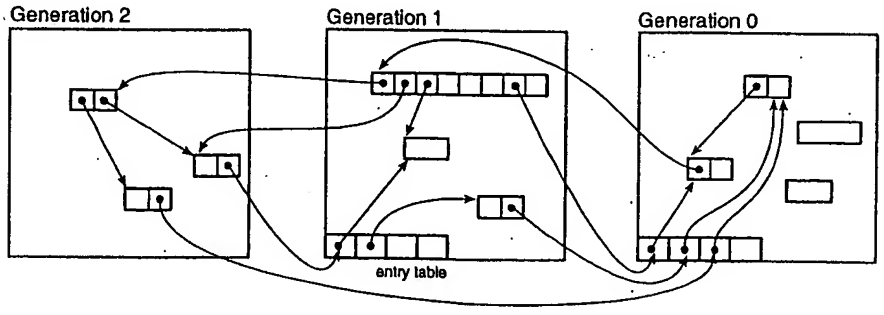

Diagram 7.17 Lieberman–Hewitt entry tables.

The advantage of this scheme is that when a younger generation is collected, it is only necessary to scavenge its entry table rather than to search every older generation. The TI Explorer garbage collector modified this scheme by maintaining a separate table for each pair of generations [Johnson, 1991a; Explorer, 1987, 1987, Section 10]. For a multi-generational collector this simplifies scanning further as only entries relevant to the generation being collected are be examined by the garbage collector.

However, indirection schemes suffer from a number of disadvantages. Entry tables may contain duplicate references to a single object, making the cost of scanning tables proportional to the number of store operations rather than simply to the number of inter-generational pointers. Trapping pointer stores and following indirections in the Lieberman and Hewitt collector would have been prohibitively expensive if the MIT Lisp Machine's specialised hardware and microcode had not made these operations invisible to the user program [Greenblatt, 1984]. Most modern generational garbage collection schemes therefore allow pointers to be used freely, referring directly to their targets. Rather than representing old–young pointers by indirections and recording the value of the pointer, these schemes record the location of the pointer.

Remembered sets

Ungar's *Generation Scavenging Collector* recorded objects that contained pointers to younger generations [Ungar, 1984]. The write-barrier, implemented in software, intercepted stores to check (a) whether a pointer was being stored, and (b) whether a reference to a young generation object was being stored into an old object. If so, the address of the old object that was to contain the pointer was added to a *remembered set* (see Diagram 7.18 on the next page). This contrasts with Lieberman and Hewitt entry tables which record pointed-to objects. To avoid duplicates in the remembered set, each object had a bit in its header indicating whether the object was already a member of the remembered set.

Scanning costs at collection time were therefore dependent on the volume of remembered set objects, rather than on the number of pointer stores. Nevertheless, the cost of store checking was high, although it was easily accommodated within interpreted Smalltalk. Worse still, if an old object were stored into several times between collections, these checks would be repeated. If the object were large, it would have to be scanned in its entirety at collection time, as the remembered set recorded the location of the object stored into rather than the location of the pointer — scanning large objects has been observed to thrash Tektronix Smalltalk [Wilson, 1994].

Collection-time scanning costs could be removed if the address of the slot within an object were remembered rather than the address of the object itself. Slot recording causes two other problems, both of which increase the size of the remembered set. Firstly, it would be impossible to avoid duplicate entries in the remembered set unless there is room to store a remembered-bit in each inter-generational pointer. Secondly, the remembered set would have to include multiple entries for a large object that had had different slots modified. We will now examine various approaches that have been used to reduce write barrier costs while also limiting the space and collection overheads that must be incurred.

Appel uses a simple and fast implementation of remembered sets [Appel, 1989b]. After every assignment into a record, instructions are emitted by the compiler to add the stored-

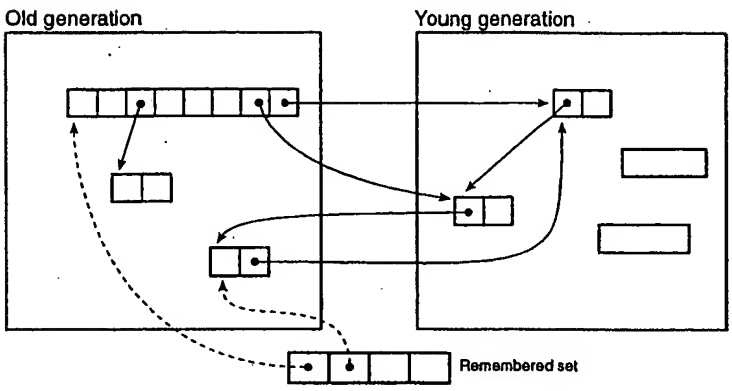

Diagram 7.18 Ungar's remembered set.

into record to an assignment list. This list need not necessarily be stored as a linked list: a contiguous vector of addresses could be used. If the vector were to overflow, either more space could be allocated for the list, not necessarily adjacent to the rest of the list, or the garbage collector could be invoked. The write-barrier is fast and unconditional: no tests that the stored value is indeed an inter-generational pointer, nor that the assignment list does not already contain the stored-into record, are used. Instead, the garbage collector filters the list, scavenging only those objects that meet these criteria. The collection-time cost of Appel's method therefore depends again on the number of pointer stores rather than on the number of objects stored into. The costs of the write-barrier and the collection-time tests were each about four instructions on a VAX. As the dynamic frequency of pointer stores for SML/NJ is less than 1 percent, the overheads caused by garbage collection as a whole are between 5 and 30 percent. This technique benefits from two features of SML/NJ. As a strict, mostly functional language, pointer stores other than initialising ones are comparatively rare. Secondly, the size of the remembered set is further reduced by the policy of *en masse* promotion at each minor collection: after each minor collection all the entries in the list can be discarded. With more generations or a different promotion policy, the list would have to be pruned and retained for use by future collections. Without these considerations, a more complex mechanism would undoubtedly be required.

Sequential Store Buffers

Hudson and Diwan maintain the remembered set for Modula-3 and Smalltalk in a way that similarly minimises the processing cost of a store, yet removes duplicate entries from the remembered sets [Hudson and Diwan, 1990; Hosking *et al.*, 1992]. A fixed-size *Sequential Store Buffer* (SSB) is again filled with addresses that might contain pointers to younger generations. The write-barrier unconditionally adds these addresses to the end of the SSB, and a 'no access' guard page is used to trap overflow. If the pointer to the next free slot in the

buffer is kept in a register, adding a word to the SSB can be done in just two additional instructions, one to store the word and the other to increment the pointer (see the code fragment in Algorithm 7.4).

```
st  %ptr, [%obj]          —store ptr into obj
st  %obj, [%ssb]          —add obj to SSB
add %ssb, 4, %ssb
```

Algorithm 7.4 The write-barrier for a sequential store buffer.

The remembered sets themselves are built as circular hash tables using linear hashing, with $2^i + k$ entries. Items to be entered are hashed to obtain i bits to index the table. If that location already contains another item, the next k slots are examined (but not circularly). If an empty slot still cannot be found, a circular search of the table is made. Hash tables are kept relatively sparse by growing them whenever an item cannot be placed in its natural slot or the next k slots, and more than 60 percent of the table is full. Hudson and Diwan set k to 2, and incremented i by one each time that it was necessary to grow the table (i.e. the table size was doubled).

If the SSB overflows, values in the SSB are moved to the remembered set of the youngest generation. Notice that hashing prevents duplicate entries from being introduced from the SSB; other uninteresting values are filtered out in a tight loop. At collection-time, entries in the SSB are similarly distributed into the appropriate remembered sets. Any values placed in the remembered set of the youngest generation when the SSB overflowed are moved to the remembered sets of the generations to which they belong. The SSB system has the advantage of a fast write trap and precision of recording, but it must still expend effort ensuring that duplicate addresses are not added to the remembered sets.

Page marking with hardware support

The CPU cost of trapping pointer stores can be reduced to nothing with specialised hardware. The Symbolics 3600 made extensive use of hardware to support both incremental and generational garbage collection. Each word stored was examined for references by the same hardware that implemented the read-barrier trap for its Baker-style incremental garbage collector (see Chapter 8 where we discuss incremental garbage collection). Any references found were stored in one of two tables (see below), but rather than recording addresses of objects, the entries referred to small virtual memory pages. This medium-level of granularity solves the problem of scanning very large objects although it increases costs if small objects are sparsely written to, as the entire page must still be scanned. At collection-time, the scavenger searched those pages recorded in the tables linearly to find references. Three features of the 3600 architecture made this technique feasible. Firstly, the Symbolics hardware write-barrier ignored any word that was not a pointer to generational data. Secondly, its tagged architecture removed the need to consider object boundaries while searching the page: pointer words could always be distinguished from non-pointer words. Finally, the pages were small — only 256 words — so a page could be scanned rapidly (in about 85 microseconds, for the 1.2 mips 3600 system, if no references were found).

Whenever a reference to generational memory (either forward or backward) was stored in any page, the write-barrier hardware set a bit in the *Garbage Collector Page Table* (GCPT) of the corresponding page-frame of physical memory. This method also had the advantage of preventing duplicates — however many times the bit is set, the page will only be searched once. Swapped-out pages were handled differently. Although swapped-in pages can be searched rapidly for pointers to the generation being collected, the cost of swapping a page in from disk, only to search it unsuccessfully, is too high. Details of swapped-out pages were held in the *Ephemeral Space Reference Table* (ESRT), a B*-tree maintained by software in non-pageable memory. The ESRT contained a bit-mask for each page, with one bit for each generation referenced by that page. When a page was evicted, its GCPT bit was cleared and the page was scanned for references, updating its ESRT table if necessary. If the page had not been written at all, its ESRT entry need not be changed. If the page had no ESRT entry, it only need be scanned if its GCPT bit was set. Otherwise the ESRT had to be updated regardless of the GCPT bit, since it was possible that data written to it might have overwritten pointers to generational objects.

Garbage collection was initiated by filling the youngest generation. All generations that were sufficiently full were flipped and a single pass was made through the GCPT, scavenging each page whose bit is set. A similar pass was made through the ESRT to complete the scavenging of the entire root set, only swapping pages in to search them if the ESRT bit for the generation being collected was set. Scavenging then continued in the normal way (although the scavenger used the 'approximately depth-first' search technique described on page 139, rather than breadth-first search, to improve locality). An advantage of this mechanism is that the Symbolics garbage collector could collect generations independently of each other, since the hardware recorded any pointer into any generation, regardless of its direction. Most other generational garbage collectors must collect all younger generations when they collect an older one.

Page marking with virtual memory support

Ephemeral Garbage Collection relied on the 3600's specialised hardware for performance. Although this is not available on stock machines, virtual memory machinery may be. Virtual memory systems use hardware to maintain a set of *dirty bits*, one for each page-frame in memory, that determine whether a page needs to be written back to disk when it is evicted. It might seem that these bits could be used as the GCPT to determine whether any pointers have been stored on a page; as far as the garbage collection system is concerned, this costs nothing. However, the situation is slightly more complex than this. A copying collector only needs to scan those pages that were written during or since the last garbage collection. We shall call this period the garbage collection interval. Shaw uses three dirty bits per resident page to keep account of modified pages [Shaw, 1988]. The `Dirty` bit, maintained by hardware, is used to track modifications made to pages since the start of the current interval. However, the virtual memory system needs to know about the state of resident pages before the interval began as well as during it — the `Old_Dirty` bit is used for this. Thus all virtual memory reads of dirtiness information must read the disjunction, `Dirty V Old_Dirty`. The `Dirty_on_Disc` bit is used for pages that have been swapped out. This system requires that the virtual memory mechanism provides two new system calls. One is a request to clear all

dirty bits for the pages of the process making the call. The other returns a map indicating which pages of the process have been written in the last interval. At collection time the garbage collector uses this map to search dirty pages for inter-generational references, having first cleared all this process's dirty bits.

However there are two problems with this approach. The virtual memory mechanism must be intercepted in order to determine which page needs scanning before it is swapped out. This may not always be possible, and is certainly operating system specific (although Shaw claimed this was very easy to do: only seventeen statements in the operating system kernel needed to be changed in addition to providing the new system calls). As an alternative to modifying the operating system kernel, pages can be write-protected by a system call. Any resulting write-faults will be trapped, and the trap handler can set a dirty bit for the page, before unprotecting it so that no further faults on this page will occur until after the next garbage collection. This technique has been used on the Xerox Portable Common Runtime system [Boehm *et al.*, 1991]. Clearly it replaces a free mechanism with one of some cost: on a SPARCStation 2, catching the Unix signal and executing the system call to unprotect the page takes about half a millisecond. However, the trap is taken at most once per page in every garbage collection cycle, rather than on every access. Boehm suggests that the virtual memory barrier can perform better than the software barrier provided that allocation rates are very low and read/write rates are moderate[9]. Reliance on virtual memory protection mechanisms makes this method unsuitable for applications with hard real-time demands.

A second problem with virtual memory based methods is that they provide a coarse write-barrier. Pages in modern systems tend to be much larger than those in the Symbolics 3600, and the virtual memory dirty bits record any modification to the page, not simply generational pointer stores. Both of these factors increase the costs of scanning a page for inter-generational references, particularly if writes are sparse.

Card marking

An ideal solution would be one that has the cheapness of a write-barrier and the economy of pointer recording of the Ephemeral Garbage Collector, but is portable and available on stock hardware. Sobalvarro proposed two methods of implementing Moon's collector for Lucid Common Lisp [Sobalvarro, 1988]. *Word marking* divides the address space into large pieces called *segments* of, say, 64 kilobytes. A *Modification Bit Table* (MBT) is associated with each segment to save space; segments which do not require pointer recording, such as those in the youngest generation, unscanned segments, and those that only contain non-pointer data, share a single MBT. Sobalvarro's MBTs occupied some 3 percent of allocated storage. When a location is modified, the bit in the MBT corresponding to that word is set unconditionally. Checking for pointers to generational data is deferred to collection time. Since the location of these modified words is recorded exactly, it is not necessary to scan segments to find them. To save the collector having to examine the MBT for each segment that might have been modified, a second-level data structure, the *segment modification cache*, is used. A byte of this cache is set non-zero whenever an entry is made in its corresponding MBT.

The cost of this write-barrier was not cheap: it used ten instructions, an address register

[9] Hans Boehm, personal communication.

and two data registers on the Motorola MC68020. A routine of this size cannot be used inline without significantly increasing the size of the program image. It is also important to restrict the number of registers needed: excessive use of these precious resources by an inline write-barrier will have a deleterious effect on the register allocation of the surrounding code.

A compromise between marking virtual memory pages and marking words, suggested by Sobalvarro, is to divide the heap into small regions called cards. *Card marking* offers several advantages provided that the cards are of the 'right' size. As they are smaller than pages, the amount of collection-time scanning is reduced. On the other hand, the amount of space a card table occupies is less than that used for word marking. Card marking is also portable and independent of the virtual memory system (although cards should not span virtual memory pages). It is also flexible since card sizes can be picked to optimise locality of reference, and to avoid allowing single stores to cause thousands of locations to be scanned at collection-time. As with word marking, a bit is set unconditionally in a card table whenever a word in the card is modified. The *Opportunistic Garbage Collector* uses a smaller card size (32 four-byte words) than either Moon's pages or Sobalvarro's segments [Wilson and Moher, 1989a; Wilson and Moher, 1989b]. Wilson and Moher argue that this size is closer to the average size of objects in Lisp or Smalltalk (excluding *cons* cells which are unlikely to be modified). By making the size of a card similar to the size of the object likely to be guilty of dirtying it, there is less room left on the card for innocent bystanders. Thus fewer objects should need scanning at collection time.

Bit manipulations usually require several instructions on modern RISC processors. This is why Sobalvarro marked bytes in the segment modification table. Using bytes rather than bits speeds up the write-barrier, reducing it to just three SPARC instructions in addition to the actual store (see the code fragment in Algorithm 7.5) [Chambers, 1992].

```
st    %ptr, [%obj + offset]          —store ptr into obj's field
add   %obj, offset, %temp            —calculate address of updated word
srl   %temp, k, %temp                —divide by card size, 2^k
clrb  [%byte_map + %temp]            —clear byte in byte_map
```

Algorithm 7.5 Chambers's write-barrier. k is $\log_2$ (card size).

The memory overhead is fairly small: with a 128-byte card, a byte map is still less than 1 percent of the heap. The cost of the barrier can be reduced still further if the accuracy of card marking is reduced. Hölzle has suggested a method of reducing the cost of the write-barrier to just two SPARC instructions in most cases, at a slight increase in scanning costs, by relaxing the accuracy with which cards are marked (see the code fragment in Algorithm 7.6 on the facing page) [Hölzle, 1993]. If byte i marked in the card table means that any card in the range $i \ldots i + l$ may contain a pointer, the byte marked may be up to l bytes from the correct one. Provided that the offset of the updated word is less than $l * 2^k$ bytes (i.e. less than l cards) from the beginning of the object, the byte corresponding to the object's address can be marked instead. A leeway of one ($l = 1$) is likely to be sufficient to cover most stores except those into array elements: these must be marked exactly in the usual way. With a 128-byte card, any field of a 32-word object can be handled.

Ambiguity only arises when the last object on a card extends into the next card. Although

```
st   %ptr, [%obj + offset]          —store ptr into obj's field
srl  %obj, k, %temp                 —calculate approximate byte index
clrb [%byte_map + %temp]            —clear byte in byte_map
```

Algorithm 7.6 Hölzle's write-barrier.

the object's address has been marked, a pointer could have been stored in any of the cards that the object straddles. This means that the garbage collector must scan the whole of the last object on a card even if only part of it belongs to the dirty card. Hölzle's figures for the SELF system on a SPARCStation 2 suggest a total garbage collection overhead of between 5 and 10 percent. In all cases, the cost of scanning cards is a fraction of the costs of store checking or scavenging.

Card marking collectors must scan dirty cards for inter-generational pointers at collection time. If none are found, the dirty bit (or byte) is cleared in the card table. The cost of scanning is proportional to the number and size of cards marked rather than the number of stores performed since duplicates never arise. Dirtiness information can also be used by the garbage collector to segregate objects on written-to cards from clean ones. By gathering dirty cards onto the same virtual memory pages, the number of pages holding cards to be scanned, and likely to be scanned again at the next scavenge, can be reduced [Wilson and Moher, 1989a].

The small size of cards presents a problem when scanning them. The tagged architecture of the 3600 allowed it to discriminate between pointer words and other data, but this facility is not available on stock hardware. Nevertheless, card marking requires that it is possible to scan a card accurately for pointers, even if the card does not start with the beginning of an object. The Opportunistic Garbage Collector uses a *crossing map* similar to that of the incremental collector described in [Appel *et al.*, 1988] (see Chapter 8). This bit- (or byte-) map, which is the same size as the card table, indicates those cards that cannot be scanned from the beginning. Cards are only scannable if they begin with the header of an object, or in the midst of an object whose subsequent data fields are tagged. If a card is unscannable, the garbage collector must skip back through the crossing map until it finds a scannable one. Page faults caused by skipping back to an earlier card from which to start the scan are undesirable. If 32-word cards are used, a 4-kilobyte page will hold 32 cards. This gives, on average, a choice of 15 cards on which to start the scan without risking a page fault. Larger card sizes would increase this risk, but smaller sizes would increase the size of the card table.

The chance of a card being scannable is also increased if large unscannable objects are stored separately in a large object area. Headerless, unscannable objects, such as floating point numbers (often represented by a tagged pointer to a one or two word value in the heap), also cause problems. These can be handled specially if all headerless objects are required to be entirely scannable or entirely unscannable. Unscannable headerless objects can then be allocated in 'containers', pseudo-objects in the heap which have a header indicating that they contain unscannable data. When a container becomes full, a fresh one is acquired from the storage manager, and new headerless unscannable objects are allocated from within the new container.

Remembered sets or cards?

For general purpose hardware, two systems look the most promising: *remembered sets with sequential store buffers* and *card marking*. Although the write-barrier costs are about the same — two instructions — in both systems, card marking provides a more predictable write-barrier overhead since the SSB may overflow. Remembered sets offer precision of pointer recording, but allow duplicates in the sequential store buffer. Processing effort at collection time is proportional to the number of stores performed between scavenges rather than the volume of data modified; this and the size of the SSB might be large. On the other hand, cards that contain inter-generational pointers remain dirty and hence have to be searched again even if they are not modified again. Hosking and Hudson took the best of both systems to provide a hybrid card marking/remembered set garbage collector for a high-performance Smalltalk interpreter [Hosking and Hudson, 1993]. The write-barrier uses card marking, but older–younger pointers are summarised to the appropriate remembered set at collection time. The remembered set is then used as the basis of the scavenge and the cards are cleaned. The write-barrier is predictable since the card table cannot overflow; no duplicates are recorded. At the cost of storing a remembered set for each generation as well as the card table, card scanning time is reduced as only those cards dirtied since the last collection need to be scanned. It would also be feasible for such a hybrid system to switch to pure card marking if the remembered sets grew excessively large. Hosking and Hudson found the hybrid scheme offered a significant improvement over pure remembered sets, with the optimal card size found to be one kilobyte. They also found that, even using sympathetic assumptions, virtual memory techniques were unlikely to be competitive, though there are other reasons (such as uncooperative compilers) that may mandate its use.

7.6 Non-copying generational garbage collection

So far we have assumed that generational garbage collectors are based on copying garbage collection. Although copy-based collectors are conceptually simpler, it is quite possible to build mark-sweep based generational collectors. Zorn examined the trade-offs between promotion threshold size, garbage collection overhead and pause length for generational garbage collection based on both stop-and-copy and mark-sweep, and concluded that his mark-and-deferred-sweep generational collector performed significantly better, for a range of substantial Allegro Common Lisp programs running on a Sun 4/280, than his copying collectors [Zorn, 1993].

Zorn's system used four generations, each of which contained a mark bitmap (see page 92), a fixed-size-object region and a variable-sized-object region. The fixed-size-object region was divided into a number of areas, each of which contained objects of a single size. These regions used mark-and-deferred-sweep garbage collection, with *en masse* promotion by copying after three collections. The variable-sized-object region contained objects that did not fit in any of the fixed-size-object areas, and was collected with a two-space copying collector. Zorn found that, although the total CPU overhead of the mark-sweep collector was slightly greater than that of the copying collector, it required 30 to 40 percent less real memory to achieve the same page-fault rate.

Mark-sweep collection also often showed a lower cache-miss rate, although this depended on promotion policy and cache size. The compacting effect of copying collection gave no advantage provided that the new space resided entirely in memory. The drawback of the mark-sweep collector was that *en masse* promotion led to much higher promotion rates; collection-count promotion would be possible if a few bits per object were reserved to record object ages (maybe in a table to the side of the heap like the mark bitmap). Increasing thresholds above one megabyte, however, led to noticeable pauses whilst thresholds less than 250 kilobytes caused increased overhead and poor locality as objects were moved out of the creation area prematurely. Zorn's results are in keeping with those of Demers *et al.* for a conservative, mark-and-deferred-sweep, generational collector for Ibuki Common Lisp on the Xerox PCR [Demers *et al.*, 1990]. Although the CPU overhead was much greater than for a non-generational collector, an order of magnitude fewer pages were touched by the generational collector.

There is no reason why all generations should be collected in the same way. In particular the oldest generations may have to be treated differently to younger ones. If a copying collector is used throughout, the oldest generation must be organised into semi-spaces since there is no older generation into which scavenge survivors can be promoted. If the cost of two semi-spaces is too high, then the oldest generation must be handled with a non-copying collector, or maybe not collected at all. If mark-sweep is used, it may occasionally be worth compacting the generation as well, especially if this can be done without paging. Ungar and Jackson built a twin-track garbage collector for PARCPlace Smalltalk-80, Release 4 [Ungar and Jackson, 1991]. In this system, most objects are reclaimed by a generational copying collector, for its efficiency and its non-disruptiveness. Compaction, which gives fast allocation for high bandwidth objects, is provided at no cost. Tenured objects, on the other hand, can be dealt with at a more leisurely rate: the key requirement is that older generations should never become so full that a major collection is needed. An incremental mark-sweep collector is used to reclaim tenured garbage. Although a first-fit (or best-fit) allocator for old objects is slower than simply incrementing a free pointer, its performance is still acceptable since objects become old at a much lower bandwidth than new ones are allocated.

7.7 Scheduling garbage collections

One of the aims of generational garbage collection is to reduce pause times. As well as concentrating on those objects most likely to be garbage, it may also be worth considering when to schedule garbage collection. Two possible strategies are either to *hide* collections at points where the user is least likely to notice pauses, or to trigger *efficient* collections when there is likely to be most garbage to collect. One way of hiding pauses in long lived systems is to arrange that major collections happen overnight, or when the machine is idle. Alternatively garbage collections can be performed at points in the program where the pauses are least likely to be disruptive. Two candidates for this are during compute-bound periods and when the user is presented with an opportunity to interact but does not do so [Wilson and Moher, 1989b; Wilson, 1990]. If garbage collection phases are attached to the end of much larger compute-bound phases, they may not exacerbate those pauses excessively. Furthermore, by

garbage collecting then, much more disturbing interruptions during interactive phases may be avoided.

There may also be points at which a program expects the user to interact, but they do not do so. If the user does not interact for a few minutes then it is probably safe to initiate a short collection. If user inactivity continues, it may be an opportune moment for a more major collection. Wilson advocates incorporating these heuristics into interactive programs by attaching code to user interaction routines. Whenever a significant pause is detected, the system can decide whether to garbage collect and if so, how many generations to scavenge. The Emacs text editor system uses a variant of this strategy: if sufficient idle time has passed, the file is auto-saved and a garbage collection is performed if sufficient allocation has been done since the last collection.

Garbage collection will be made more efficient if it is run at times when the volume of live data is low. The ends of compute-bound periods or user interaction points may also of themselves be good times to collect, since they are often dispatching points between major computations. It is quite likely that the volume of live data is low at these times. Other opportune moments include at the local minima of stack height [Lieberman and Hewitt, 1983], or when the number of page-faults becomes excessive. In the latter case, the goals of the collection are to compact data to improve locality of reference [Wilson *et al.*, 1991]. The garbage collector itself may also be able to detect likely opportunities. If the number of objects reclaimed during the last scavenge of a younger generation was high, it may be worth scavenging its older neighbour as well [Hudson and Diwan, 1990].

Detecting true local minima of the stack height is problematic. One approximate solution is to trigger a collection whenever the stack height drops below a certain point. Wilson suggests that one method may be to place a bogus return address in the stack [Wilson, 1991]. If this is used as a return address, control can be passed to a routine that determines whether it is worth calling the collector. This decision may be based on the amount of memory currently available, and the height of the stack. Wilson reports that the success of this strategy is application dependent. Where it is successful, it tends primarily to decrease the amount of data copied at the first scavenge but the proportion of data that survives two scavenges increases slightly.

Key objects

Hayes observed that the deaths of objects allocated at nearly the same time are closely correlated [Hayes, 1991]. He observed the behaviour of the 1 percent of objects with the longest lifetimes and found that more than 60 percent of these words came from a spread of ± 1 kilobyte. If young objects were included, the correlation was unsurprisingly much stronger. These object demographics arise from typical styles of programming. There are usually only a few static pointers into large data structures: an example would be the root of a tree. Other pointers into the structure are created dynamically as it is used by the program. When the program has finished with the tree, it will only be reachable by its root. When this pointer is deleted, the entire tree is garbage.

Hayes suggests using these *key* objects as indicators for garbage collections. When the death of a cluster of objects can no longer be predicted from their age, the cluster should be promoted out of the time-based generational scheme altogether: no effort should be made to collect it (see Diagrams 7.19 and 7.20). One key object, for example the root of a tree

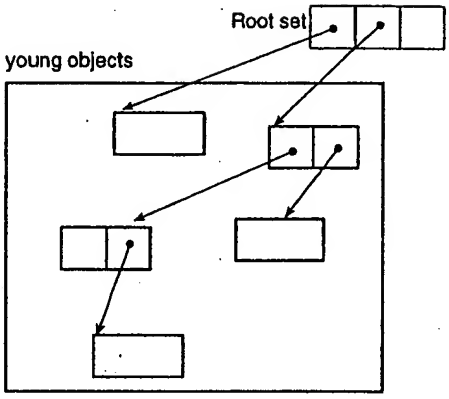

Diagram 7.19 Key objects: before promotion.

or the head of a list, should be retained within the generational scheme. The reachability of this key object is then used to suggest when to collect the cluster. Reclaiming a key object is interpreted as a hint that it might be worth trying to reclaim the keyed cluster associated with it. Collecting older generations along with all young generations in a large heap without disruption is problematic. Avoiding collecting objects unless there is a good reason to think that the attempt might be successful is a sound strategy, if it can be implemented.

Mature object spaces

The difficulty with this scheme is how key objects are to be identified. One method is a manual one, with the programmer offering hints on which objects are thought to be good predictors. Alternatively, if a cluster was accessible from the stack, these direct references could serve as keys. An added bonus is that this technique would also indicate that the stack had shrunk without explicitly monitoring it. Hudson and Moss describe a further mechanism, inspired by key object opportunism, that collects clusters of objects by detecting when the cluster is unreachable [Hudson and Moss, 1992]. Like Hayes, they promote very old objects out of the time-based generational scheme altogether and into a *mature object space* in order to avoid disruptive collections. Their design is similar to the distributed algorithm described in [Shapiro *et al.*, 1990]. The mature object space is divided into *areas*, each of which has a remembered set. These are collected one at a time, in a round-robin fashion, thus placing an upper bound on the length of any collection. An area is reclaimed in its entirety when its remembered set is empty, i.e. when there are no references to objects in the area from outside the area.

A difficulty arises if a cluster of linked objects is too large to fit in a single area, since areas are bounded in size (unlike the areas described in [Bishop, 1977]). Hudson and Moss use a train analogy to describe their solution to this problem, with *carriages* representing areas, and *trains* representing groups of carriages holding linked structures. At each collection, a single

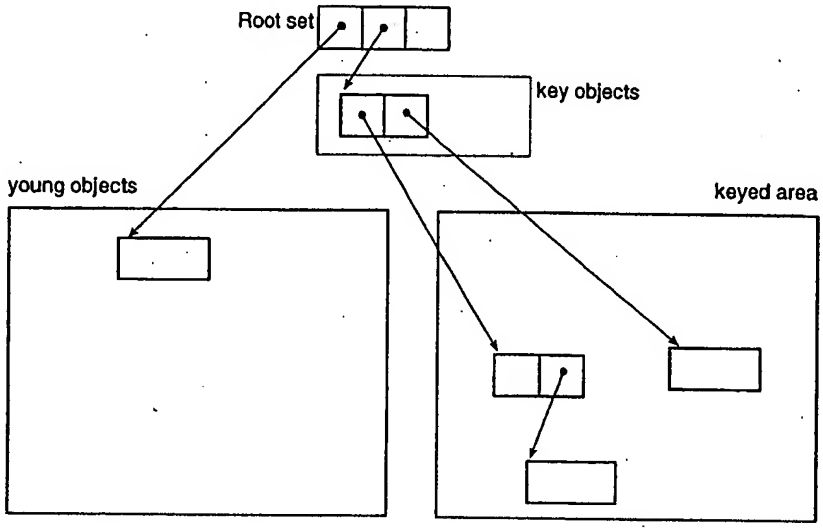

Diagram 7.20 Key objects promoted out of the generational scheme.

carriage is chosen for collection: call it the *From-carriage*. If there are no references to the From-carriage's train from outside that train, the entire train can be reclaimed. Otherwise, collection proceeds in four phases.

First, any object that is referenced from outside the mature object space is moved into a fresh train. Suppose that the top-left carriage in Diagram 7.21 on the next page is the From-carriage. The only external reference is to object A; A is copied to a new train (see Diagram 7.22 on page 180). These objects are then scanned in the usual copying collector way, and any descendants also in the From-carriage (for example, B in the diagrams) are moved to this new train. Promoted objects are also moved into trains in this phase. At this point, references to objects in the From-carriage are held only in mature-space objects.

In the third phase, From-carriage objects referenced from other trains in the mature object space (for example, P) are moved to those trains. Those referenced from other carriages in the From-train (for example, X) are moved to the last carriage in this train. This leaves the From-carriage containing only unreachable objects (for example, the group of three objects shown in the lower left-hand corner of the From-carriage), so the entire carriage is recycled. Once an entire structure is held in a single train, it can be reclaimed if there are no external references to it (for example, in the collection cycle that follows the state shown in Diagram 7.22 on page 180, the train holding Y and X can be reclaimed in its entirety). This system has a number of attractive features. It is incremental since the number of bytes moved at each collection is bounded. Objects are clustered and compacted as they are copied into cars. The system is efficient in that it does not rely on special hardware or virtual memory mechanisms.

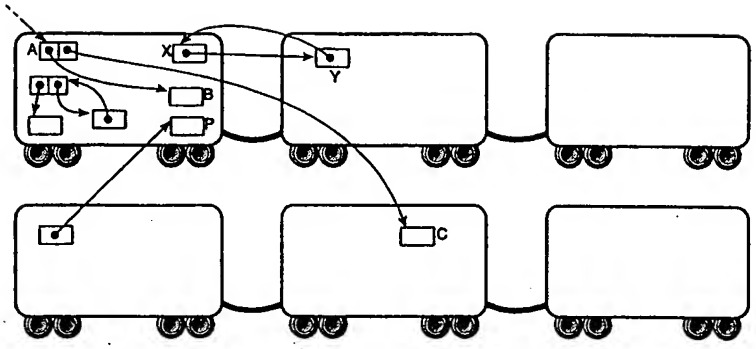

Diagram 7.21 The mature object space before the From-carriage is collected. Only details of interesting carriages are shown.

7.8 Issues to consider

Generational garbage collection has proved to be highly successful in a wide range of applications. It can reduce pause times for minor collections to a level where it is worth considering instead of incremental techniques for some applications. By concentrating allocation and collection effort on a smaller region of the heap, paging and cache behaviour of both the mutator and the collector can often be improved. Finally, by delaying collection of long-lived objects, generational techniques can reduce the overall cost of garbage collection. Programming styles which allocate large numbers of short-lived objects, and in which non-initialising pointer writes are comparatively rare, benefit particularly from this approach. However, generational garbage collection is not a universal panacea and certain circumstances may not satisfy the weak generational hypothesis.

The goal of short pause times is defeated by large root sets. These may be caused by any combination of very large programs, an unusually large number of global variables pointing into the heap, or highly recursive calls leading to very deep stacks. One solution to the problem of large stacks is to apply the write-barrier to local variables as well, although this would considerably increase the cost of the barrier.

Alternatively, if objects are promoted to the next generation *en masse* at every minor collection, it is not necessary for the collector to scan every stack frame. In this case, the only activation records that can contain old–young pointers are those created since the last allocation. All that is necessary is that the collector mark the top frame of the stack. At the next minor collection only frames above this one need to be scanned for pointers into the young generation. The only difficulty with this approach is that the 'high water mark' frame might be popped between collections. Appel and Shao suggest that the frame be marked by replacing its return address with the return address of a 'mark-shifting' procedure [Appel and Shao, 1994]. When the frame is popped, control will pass to this procedure, which will mark the next-lower frame in the same way before jumping to the real return address. Appel and Shao estimate the cost of handling high water marks at between 10 and 100 instructions.